# Part 1: Setup of GNURadio and Pluto SDR

* Install GNU Radio from the following link and select according to system requirements
  + <https://github.com/ryanvolz/radioconda>

A screenshot of a computer

AI-generated content may be incorrect.

# Part 2: Install Pluto SDR drivers

* Download and install the drivers for Pluto SDR. (<https://wiki.analog.com/university/tools/pluto/users/quick_start>) Click on the link for your OS as indicated in the red box.

A screenshot of a computer

AI-generated content may be incorrect.

* For Windows, download the driver as indicated by the red box below. Double click on the downloaded package and follow instructions to install. For other OS, refer to instructions on website.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

# Part 3: Running existing source code

* Select File->Open , select. grc folder where it was downloaded

A screenshot of a computer

AI-generated content may be incorrect.

# Part 4: Fix COP execute error

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer error message

AI-generated content may be incorrect.

* Walkthrough
  + Disable COP Execute block and the connected Hexdump Sink block
  + Run the GNURadio file. The location where the GNURadio file is saved in should have additional Python files loaded
  + Stop and close GNURadio file
  + (On Windows) Right-click on “This PC” or “My Computer” and select “Properties”.
  + Click on “Advanced system settings”
  + Click on “Environment Variables”
  + Under “System Variables”, click “New” and add a variable named PYTHONPATH and the desired directory as its value
  + Reopen GNURadio and enable back COP Execute and Hexdump Sink, the error should disappear

A screenshot of a computer

AI-generated content may be incorrect.

\*must delete and add path again after adding extra python modules

\*must close GNURadio before adding path

# Part 5: Adding new PLUTO SDRs

* Step 1: Connect both PLUTOs on the same device
  + (NOTE: use different IP addresses)
  + (NOTE: When connecting on same device, use different subnet)
* Step 2: Identify PlutoSDR -> Left click -> Locate config file

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

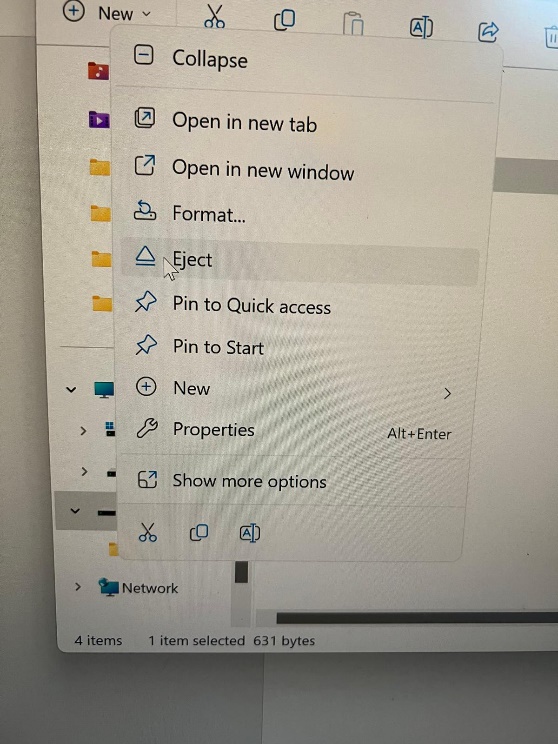
AI-generated content may be incorrect.

* Step 3: Edit parameters

A screenshot of a computer

AI-generated content may be incorrect.

* Step 4: Eject device by right clicking on PLUTO tab



# Part 6: Using single PLUTO

In the scenario where a single PLUTO is important to note that both the telecommand and telemetry cannot run concurrently. Hence, the following steps should be taken.

To run only telecommand, firstly focus on the left-half of the design which are the telecommand blocks for ground and space.

Perform the following steps:

* Turn on Pluto Sink and Source (Green)
* Turn off noise source blocks as shown in the picture below (Red)

A computer screen shot of a diagram

AI-generated content may be incorrect.

* Now head over to the telemetry portion of the design, right-half. Perform the following steps
  + Turn on noise channel blocks (Green)
  + Turn off PLUTO SDR sink and Source (Red)

A computer screen shot of a diagram

AI-generated content may be incorrect.

To run only telemetry portion of the design the approach is similar. This time start from the telemetry blocks where the following steps has to be carried out.

* Turn on PLUTO Sink and Source (green)
* Turn off noise channel (red)

A computer screen shot of a computer

AI-generated content may be incorrect.

* Head over to the telecommand portion to
  + Turn on noise channel
  + Turn off Pluto SDR Sink and Source

A computer screen shot of a computer

AI-generated content may be incorrect.

# Part 7: Overall architecture

1. **Telecommand ground side**

A diagram of a computer network

AI-generated content may be incorrect.

|  |  |
| --- | --- |
| **Key components** | **Functions** |
| Socket PDU | TCP Port which receives data from a python script tagging to the specified port number |
| Custom PDU Randomiser to Custom PDU Add start/stop | BCH Encoding, adding start stop sequences to telemetry messages |
| Constellation modulator | Perform BPSK modulation |
| PlutoSDR Sink | Equivalent to Tx port of PLUTO |

1. **Telecommand space side**

**A cartoon characters connected to a computer

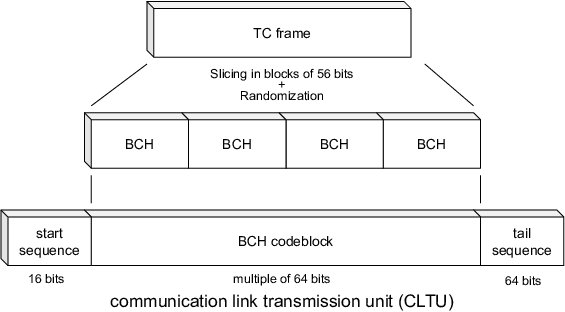
AI-generated content may be incorrect.**

|  |  |
| --- | --- |
| **Key components** | **Functions** |
| PLUTOSDR Source | Equivalent of Rx port of PLUTO |
| Polyphase clock Sync, Costas loop and constellation decoder | Decoding and demodulation |
| Correlate Access code – Tag *(NEW)* | Locate Start and Tail sequence of CLTU |
| Mock Satellite *(NEW)* | * Break into 8 byte BCH codeblocks * Add filler byte if necessary * Remove Error control bits |

**Telecommand relevant data**

A yellow box with black text

AI-generated content may be incorrect.



A diagram of a computer code

AI-generated content may be incorrect.

1. **Telemetry space side**

**A diagram of a computer

AI-generated content may be incorrect.**

|  |  |
| --- | --- |
| **Key components** | **Functions** |
| Socket PDU | Receive data through python script: telemetry tcp client |
| Intermediate blocks between Socket PDU and blk\_tm\_frame | RS Encoding , convolutional encoding |
| BLK\_TM\_FRAME (New) | Encapsulation of data into:  10 dummy+ Actual + 10 dummy data |
| Constellation modulator | BPSK modulation |
| PlutoSDR Sink | Equivalent to Tx port of PLUTO |

1. **Telemetry ground side**

**A computer screen shot of a computer

AI-generated content may be incorrect.**

|  |  |
| --- | --- |
| PLUTOSDR Source | Equivalent of Rx port of PLUTO |
| Polyphase clock Sync, Costas loop and constellation soft decoder | Decoding and demodulation |
| CCSDS Concatenated deframer | Handle decoder outputs and feed to COP execute again |

# Part 8: Overview of files

A screenshot of a computer

AI-generated content may be incorrect.

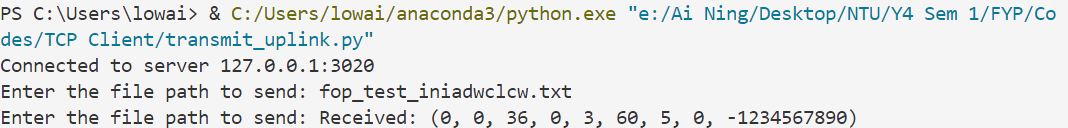
* COP Requests contains the binary files that contains test directives/requests/CLCWs to send to the GNURadio flowgraph to test the COP.
* Generate COP requests is the Python code that I use to generate the COP requests binary files.
* GNURadio contains the Python codes that is implemented in the Python blocks/modules in the GNURadio flowgraph. They have the same name as their blocks in the GNURadio flowgraph.
* TCP Client has 5 files as seen below.
  + Check\_downlink\_output.py is to compare the data sent and received through the downlink.
  + Duplicate\_data.py is to duplicate an encoded TM transfer frame by 20 times and save to a file
  + Receive\_downlink.py is a TCP client that connects to the TCP server in GNURadio to receive decoded downlink TM transfer frames and save to a file.
  + Transmit\_uplink.py is a TCP client that connect to the TCP server in GNURadio to send directives/data to transmit through the uplink and receive and print out responses to the directives.
  + Transmit\_clcw\_uplink.py is a TCP client that connect to the TCP server in GNURadio to send CLCWs to the COP in the uplink. It is to mimick the frames received from the downlink, which the COP will check in order to obtain and check the CLCWs.

A number and time on a white background

AI-generated content may be incorrect.

# Part 9: COP Execute details

* To test the COP, enable the PDU to Tagged Stream block and disable the Vector Source block. Run the GNURadio flowgraph. Run the transmit\_uplink.py file in TCP Client folder.
* To send a directive, enter the name of the binary file containing the directives eg fop\_test\_iniadwclcw.txt (in the COP\_Requests folder) in the console to send a directive to Initialise AD (with CLCW). The response received (Accept) is printed out. In the GNURadio, the requests received and the directives executed are also shown in the console.

A screenshot of a computer

AI-generated content may be incorrect.

* To send a clcw, run the transmit\_clcw\_uplink.py file in the TCP Client folder. Enter the name of the binary file containing the clcw eg fop\_test\_clcw0.txt (in the COP\_Requests folder) in the console. The clcw received is shown in the GNURadio console and the response(Positive Confirm) after receiving the clcw required to complete the AD initialisation is also shown. In the transmit\_uplink.py console, the Positive Confirm response received is also printed out.

A computer screen shot of a computer

AI-generated content may be incorrect.A computer screen shot of a computer screen

AI-generated content may be incorrect.

A computer screen shot of a code

AI-generated content may be incorrect.

* To continue sending other directives, type the binary file name into the transmit\_uplink.py console eg fop\_test\_segmentad.txt to send a TC segment(AD). Type fop\_test\_clcw.txt into the transmit\_clcw\_uplink.py console to send a clcw to acknowledge the segment. The expected output in the GNURadio console is also shown below.

A computer screen shot of a code

AI-generated content may be incorrect.A computer screen shot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* Stop the flowgraph when completed.

# Part 10: COP testing

|  |  |  |
| --- | --- | --- |
| **COP Execute parameters** | **Default value set** |  |
| endianess | ‘big’/ ‘little’ | Whether the TCP messages sent from the upper layer and the TCP messages sent to the upper layer is in big/little endian  \*change the endian parameter in Sample\_directives\_requests.ipynb accordingly to generate the directives/requests in the correct endianness  \*change the endian parameter in transmit\_uplink accordingly to print the received TCP messages correctly |
| fecf\_present | False/ True | Whether frame error control field is included in TC transfer frame |
| spacecraft\_id | 80 |  |
| fop\_sliding\_window | 3 |  |
| transmission\_limit | 2 |  |
| timeout\_type | 0 |  |
| timer\_initial\_value (timeout/T1) | 100 seconds |  |

|  |  |  |
| --- | --- | --- |
| **File name** | **Message type** | **Details** |
| fop\_test\_clcw.txt | tm transfer frame containing clcw | n(r) = 1 |
| fop\_test\_clcw\_retransmit.txt | tm transfer frame containing clcw | n(r) = 0  retransmit\_flag = 1 |
| fop\_test\_clcw0.txt | tm transfer frame containing clcw | n(r) = 0 |
| fop\_test\_clcw2.txt | tm transfer frame containing clcw | n(r) = 2 |
| fop\_test\_clcw120.txt | tm transfer frame containing clcw | n(r) = 120 |
| fop\_test\_clcw121.txt | tm transfer frame containing clcw | n(r) = 121 |
| fop\_test\_iniad.txt | initiate ad without clcw directive |  |
| fop\_test\_iniadwclcw.txt | initiate ad with clcw directive |  |
| fop\_test\_iniadwsetvr.txt | initiate ad with set v(r) directive |  |
| fop\_test\_iniadwsetvr\_little.txt | initiate ad with set v(r) directive | little endian |
| fop\_test\_iniadwunlock.txt | initiate ad with unlock directive |  |
| fop\_test\_packet.txt | TC packet |  |
| fop\_test\_resumead.txt | resume ad directive |  |
| fop\_test\_segmentad.txt | ad segment transfer request |  |
| fop\_test\_segmentad\_little.txt | ad segment transfer request | little endian |
| fop\_test\_segmentbd.txt | bd segment transfer request |  |
| fop\_test\_segmentbd\_little.txt | bd segment transfer request | little endian |
| fop\_test\_sett1.txt | set T1 directive | T1 (timeout timer) = 50 (seconds) |
| fop\_test\_settimeouttype.txt | set timeout type directive | timeout type = 1 |
| fop\_test\_settxlimit.txt | set transmission limit directive | transmission limit = 5 |
| fop\_test\_settxlimit1.txt | set transmission limit directive | transmission limit = 1 |
| fop\_test\_setvs.txt | set V(S) to V\*(S) directive | V(S) = 125 |
| fop\_test\_setwindow.txt | set fop sliding window directive | sliding window = 5 |
| fop\_test\_tcframe.txt | TC Transfer Frame |  |
| fop\_test\_terminatead.txt | terminate ad directive |  |

Steps for testing COP:

Run the GNURadio file. Run transmit\_uplink.py and transmit\_clcw\_uplink.py. Enter the file to transmit according to sequence below in the correct Python file for the respective tests. Eg to Test AD transfer (with positive ack), type fop\_test\_iniadwclcw.txt in the console for transmit\_uplink.py, followed by fop\_test\_clcw0.txt in the console for transmit\_clcw\_uplink.py, then fop\_test\_segmentad.txt in the console for transmit\_uplink.py, and fop\_test\_clcw.txt in the console for transmit\_clcw\_uplink.py.

|  |  |  |
| --- | --- | --- |
|  | transmit\_uplink.py | transmit\_clcw\_uplink.py |
| Test BD transfer | fop\_test\_segmentbd.txt |  |
| Test AD transfer (with positive ack) | fop\_test\_iniadwclcw.txt  fop\_test\_segmentad.txt | fop\_test\_clcw0.txt  fop\_test\_clcw.txt |
| Test AD transfer (with negative ack ie retransmit)  \*wait\_flag = 0 | fop\_test\_iniadwclcw.txt  fop\_test\_segmentad.txt | fop\_test\_clcw0.txt  fop\_test\_clcw\_retransmit.txt |
| Test AD transfer (with auto retransmit after timeout) | fop\_test\_iniadwclcw.txt  fop\_test\_segmentad.txt  (wait for 100s) | fop\_test\_clcw0.txt |
| Initiate AD service (without CLCW) | fop\_test\_iniad.txt |  |
| Initiate AD service (with CLCW) | fop\_test\_iniadwclcw.txt |  |
| Initiate AD service (with set V(R)) and AD transfer  \*V(R) = 120 | fop\_test\_iniadwsetvr.txt  fop\_test\_segmentad.txt | fop\_test\_clcw120.txt  fop\_test\_clcw121.txt |
| Initiate AD service (with unlock) | fop\_test\_iniadwunlock.txt | fop\_test\_clcw0.txt |
| Terminate AD service | fop\_test\_iniad.txt  fop\_test\_terminatead.txt |  |
| Resume AD service | fop\_test\_settimeouttype.txt  fop\_test\_iniad.txt  fop\_test\_segmentad.txt  (wait for 100s)  fop\_test\_resumead.txt | fop\_test\_clcw\_retransmit.txt |
| Set V(S) | fop\_test\_setvs.txt |  |
| Set fop sliding window  \*sliding window = 5 | fop\_test\_setwindow.txt |  |
| Set T1  \*T1 = 50 | fop\_test\_sett1.txt |  |
| Set transmission limit  \*limit = 5 | fop\_test\_settxlimit.txt |  |
| Set timeout type  \*timeout type = 1 | fop\_test\_settimeouttype.txt |  |
| Test AD transfer (with positive ack) (little endian)  \*change COP Execute block endian parameter to ‘little’  \*change endian parameter in transmit\_uplink.py to ‘little’ | fop\_test\_iniadwsetvr\_little.txt  fop\_test\_segmentad\_little.txt | fop\_test\_clcw120.txt  fop\_test\_clcw121.txt |

# Part 10: Tips on Writing Python blocks

|  |  |
| --- | --- |
| To revise Python code | * Double click on “Embedded Python Block” * Select “Open in Editor” * Select “Use Default” or “Choose Editor” (I select “Choose Editor” 🡪 navigate to C:\Program Files (x86)\Notepad++ 🡪 Double click on notepad++.exe) * When you have finished editing, save the file (usually Ctrl-S). * close the edit window. * If there are any syntax errors, they are shown in the error window. |
| Pre populated code structure | *"""*  *Embedded Python Blocks:*  *Each time this file is saved, GRC will instantiate the first class it finds*  *to get ports and parameters of your block. The arguments to \_\_init\_\_ will*  *be the parameters. All of them are required to have default values!*  *"""*  **import** **numpy** **as** **np**  **from** **gnuradio** **import** gr  **class** **blk**(gr.sync\_block): *# other base classes are basic\_block, decim\_block, interp\_block*  *"""Embedded Python Block example - a simple multiply const"""*  **def** \_\_init\_\_(self, example\_param=1.0): *# only default arguments here*  *"""arguments to this function show up as parameters in GRC"""*  gr.sync\_block.\_\_init\_\_(  self,  name='Embedded Python Block', *# will show up in GRC*  in\_sig=[np.complex64],  out\_sig=[np.complex64]  )  *# if an attribute with the same name as a parameter is found,*  *# a callback is registered (properties work, too).*  self.example\_param = example\_param  **def** work(self, input\_items, output\_items):  *"""example: multiply with constant"""*  output\_items[0][:] = input\_items[0] \* self.example\_param  **return** len(output\_items[0])  This is the pre-populated code. It simply takes the input stream and multiplies it by a constant. |
| GNURadio block buffer in/out | Each block has input and output vectors which will hold the samples for each input/output stream |
| Create a class from GNURadio base classes | There are 4 types of GNURadio blocks   1. Synchronous block (default given in the pre populated code)  * consume and produce an equal number of items per port (1:1) * functions: \_\_init\_\_ & work * **class** **blk**(gr.sync\_block):  1. Basic block  * input and output size can be different (N:M) * functions: \_\_init\_\_ & forecast & general\_work * **class** **blk**(gr.basic\_block):  1. Interpolation block  * Input size < output size (1:M) * **class** **blk**(gr.interp\_block):  1. Decimation block  * Input size > output size (N:1) * **class** **blk**(gr.decim\_block): |
| \_\_init\_\_ function | 1. def \_\_init\_\_(self, example\_param = 1.0): 🡪 include all parameters and their default value (compulsory), their values can be changed in the GNURadio GUI 2. name 🡪 give the Python block a name (will be shown in flowgraph) 3. in\_sig 🡪 state the data type of input port(s)  * in\_sig=None would turn this into a source block (no input) * in\_sig=[numpy.float32, numpy.float32] for 2 input ports  1. out\_sig 🡪 state the data type of output port(s)  * out\_sig=None would turn this into a sink block (no output) * out\_sig=[numpy.float32, numpy.float32] for 2 output ports |
| work function | 1. define core processing behavior of block 2. input\_items and output\_items are lists of data arrays. The input\_items contains a vector of input samples for every input stream (port), and the output\_items is a vector for each output stream (port) where we can place items. Eg input\_items[0] contains input data for first port 3. return the total number of items in each output buffer |
| forecast function | A screenshot of a computer program  AI-generated content may be incorrect.   1. provide downstream blocks with information about the expected number of input items for a given number of output items. Allows us to signal to the scheduler to invoke our general\_work function only when a sufficient number of input elements are in the input buffer 2. How it works:   Eg A white background with black text and numbers  AI-generated content may be incorrect.   1. The forecast above predicts that the input buffer needed is 7 samples less than the output buffer eg for output of 32768 samples, the input buffer size must be at least 32761 2. During runtime, the scheduler asks forecast what input buffer size is needed to produce the largest amount of output that the output buffer can deal with (32768), which will be 32761 3. Scheduler checks if the input buffer has enough samples, if yes then that amount of output buffer is provided to general\_work (ie len(output\_items[0])), if no then it halves the output buffer size (16384) and repeat step 2 & 3 until it hits the minimum output buffer size (1). If none of these requests yield a fulfillable input requirement, the block is permanently input-blocked. [9] 4. After a certain output buffer size is determined by forecast, the general\_work function is called. Forecast function does not affect input\_items parameter but will determine length of output\_items[0]. Forecast only provides a prediction of output buffer size needed but the actual size of input buffer consumed and output buffer produced is still controlled by general\_work 5. noutput\_items is the output buffer size expected 6. ninputs is the number of input ports 7. It expects a return of an array with the expected number of input items for each input port |
| general\_work function | A screenshot of a computer program  AI-generated content may be incorrect.   1. define core processing behavior of block (similar to work function) 2. input\_items and output\_items are lists of lists. The input\_items contains a vector of input samples for every input stream (port), and the output\_items is a vector for each output stream (port) where we can place items. Eg input\_items[0] = input data for first port 3. Use the consume method to tell the scheduler the number of input\_items consumed for which input port 4. return the total number of output items produced |
| Creating new variables | self.variable1 = 10 |
| List of data types | numpy.int8, numpy.int16, numpy.float32, numpy.complex64 |
| **PMT (polymorphic types)** | import pmt |
| Message passing | Message passing is a mechanism to asynchronously communicate events and data between blocks. Unlike streams and tags, messages can be passed to upstream blocks. There are many uses for message passing, such as sending feedback to upstream blocks, transfer PDU (Protocol Data Unit)  The message passed can be of different types. The example here is of pdu data type  A white screen with text on it  AI-generated content may be incorrect.   1. self.message\_port\_register\_in(pmt.intern(‘msg\_in’)) 🡪 register an input message port with name ‘msg\_in’, which is in practice a PMT symbol (i.e., an interned string). 2. self.message\_port\_register\_out(pmt.intern(‘msg\_out’)) 🡪 register an output message port with name ‘msg\_out’, which is in practice a PMT symbol (i.e., an interned string). 3. self.set\_msg\_handler(pmt.intern(‘msg\_in’), self.handle\_msg) 🡪 set message handler for input port ‘msg\_in’, which indicate the function to be called when a message is received 4. pmt.cdr 🡪 cdr retrieve the 2nd element of the pmt, which is a pmt of an array of the data in the pdu 5. pmt.to\_python(pmt.cdr(msg)) 🡪 convert the pmt object to its equivalent python representation (in this case, an array with the data of the pdu is obtained) 6. self.message\_port\_pub(pmt.intern(‘msg\_out’), pmt.cons(pmt.PMT\_NIL, pmt.init\_u8vector(len(data), data)) 🡪 output the pdu to message port msg\_out. Any block that has a subscription to output message port ‘msg\_out’ (by connecting message ports as part of the flowgraph) will receive the message when it is published |
| Reading/Writing stream tags | Stream tag is a parallel stream to the data streaming. The difference is that tags are designed to hold metadata and control information. Tags are specifically associated with a particular sample in the data stream and flow downstream alongside the data. This model allows other blocks to identify that an event or action has occurred or should occur on a particular item. Tag stream is really only accessible inside a work function and only flows in one direction. [10]  A tag decorates a stream with metadata. A tag is associated with a particular item in a stream. An item may have more than one tag associated with it. The association of an item and tag is made through an absolute count. Every item in a stream has an absolute count. Tags use this count to identify which item in a stream to which they are associated.  A tag has the following members:   * **offset:** the unique item count * **key:** a PMT key unique to the type of contents * **value:** a PMT holding the contents of this tag * **srcid:** a PMT id unique to the producer of the tag (optional) [11]   Tags can be used to identify PDU boundaries in a streamed, but packetized data. Think of packet data transmission: A data packet consists of N bytes. However, in traditional GNU Radio blocks, if we stream N bytes into a block, there's no way of knowing the packet boundary. On the first item of a streamed PDU, there must be a tag with a specific key, which stores the length of the PDU as a PMT integer. [12]  A PMT is a special data type in GNURadio to serialize arbitrary data.  key = pmt.intern("packet\_len")  value = pmt.from\_long(in\_length)  self.add\_item\_tag(0, # Write to output port 0            self.nitems\_written(0), # Index of the tag in absolute terms            key, # Key of the tag            value # Value of the tag  )  A screenshot of a computer program  AI-generated content may be incorrect. |
| Other pmt functions | PMTs can represent the following:   * Boolean values of true/false * Strings (as symbols) * Integers (long and uint64) * Floats (as doubles) * Complex (as two doubles) * Pairs * Tuples * Vectors (of PMTs) * Uniform vectors (of any standard data type) * Dictionaries (list of key:value pairs) * Any (contains a boost::any pointer to hold anything) [13] * pmt.intern(“name”) 🡪 create a PMT symbol (i.e. string) * pmt.cons(KEY, VALUE) 🡪 KEY as argument name (KEYs should be pmt.symbols (i.e. strings)), VALUE as value (VALUEs can be whatever the block requires), eg msg = pmt.cons(pmt.PMT\_NIL, pmt.make\_u8vector(16, 0xFF)) in the eg, it is used to create a PDU. In GNURadio, we define a PDU as a PMT pair of (metadata, data). The PDU's metadata section is empty, hence the pmt.PMT\_NIL object. The payload is now just a simple vector of 16 bytes of all 1's. * pmt.to\_bool(msg) 🡪 takes the message PMT and then converts the data type into Python's Boolean data type [14] * pmt.from\_bool(true) 🡪 convert boolean data type to PMT [14] * pmt.from\_long(23) 🡪 convert long data type to PMT * pmt.from\_double(0.2) 🡪 convert double data type to PMT * pmt.from\_complex(1j) 🡪 convert complex data type to PMT * pmt.is\_integer(P) 🡪 return boolean whether PMT ‘P’ is integer data type * pmt.is\_real(P) 🡪 return boolean whether PMT ‘P’ is real (int/double) * pmt.is\_complex(P) 🡪 return boolean whether PMT ‘P’ is complex data type   pmt.is\_vector(P) 🡪 return boolean whether PMT ‘P’ is vector  pmt.is\_f32vector(P) 🡪 return boolean whether PMT ‘P’ is vector with elements of type float32 (u8/s8/u16/s16/u32/s32/u64/s64/f32/f64/c32/c64)   * pmt.to\_pmt(42) 🡪 make use of dynamic typing, convert any data type to PMT * pmt.to\_pmt((1,2,3,‘spam’,‘eggs’)) 🡪 tuple to PMT * pmt.to\_pmt({‘spam’:42, ‘eggs’:23}) 🡪 dictionary to PMT * pmt.to\_pmt([42,23,23,23,23]) 🡪 vector to PMT * pmt.to\_pmt(numpy.array([2.0,5.0,5.0,5.0,5.0], dtype=numpy.float32)) * pmt.to\_python(P) 🡪 make use of dynamic typing, convert PMT ‘P’ to its Python data type * a = pmt.make\_dict()   a = pmt.dict\_add(a, key0, val0) (\*key0 is PMT string, val0 is PMT int)  pmt.dict\_has\_key(a, key0)  a = pmt.dict\_delete(a, key0)   * pmt.PMT\_T 🡪 boolean PMT types representing True * pmt.PMT\_F 🡪 boolean PMT types representing False * pmt.PMT\_NIL 🡪 Null/None PMT types |